Journal of Universal Computer Science, vol. 11, no. 7 (2005), 1327-1348
submitted: 28/4/05, accepted: 9/6/05, appeared: 28/7/05 © J.UCS

An Equational Specification for the Scheme Language[1]

Marcelo d'Amorim[2] Grigore Roşu
Formal Systems Laboratory
Department of Computer Science, University of Illinois Urbana-Champaign
{damorim,grosu}@cs.uiuc.edu

Abstract: This work describes the formal semantics of SCHEME[3] as an equational theory in the MAUDE rewriting system. The semantics is based on continuations and is highly modular. We briefly investigate the relationship between our methodology for defining programming languages and other semantic formalisms. We conclude by showing some performance results of the interpreter obtained for free from the executable specification.
Key Words: formal and executable semantics, equational specification, program analysis
Category: D.3.1, D.3.4

1 Introduction

Following the approach proposed in [34] and partially published in [27] to give modular rewriting logic [25] executable semantics to programming languages, this paper describes a formal semantics of SCHEME [24]. The efficient rewriting engine of MAUDE [3] gives us an interpreter for SCHEME from its semantics essentially for free. Since SCHEME is not a concurrent language, we only need the equational fragment of rewriting logic.

The use of rewriting logic is not novel in the definition of programming languages. In fact, one can translate the structural operational semantics (SOS) [32] of a sequential language into a set of equations [27, 26] satisfying the *Church-Rosser* property. Braga *et al.* [6] investigated how to translate language definitions given in the form of Mosses' MSOS [30] into modular rewriting semantics (MRS) [8] in order to mechanically derive sound language interpreters for concurrent languages.

As in [6], our main focus is on modular definitions of languages using rewriting. In [34], we found (and empirically evaluated in class) that the use of *continuations* [4, 16], in addition to that of associative and commutative (AC) *sets of state attributes*, can improve modularity of language definitions. Using this technique, we here illustrate how easily a complex language like SCHEME can be

[1] A preliminary version of this paper appeared in the proceedings of SBLP 2005 [9]. This research is supported in part by the joint NSF/NASA grant CCR 0234524, NSF CCF-0448501, and NSF CNS-0509321.
[2] Supported by CAPES grant# 15021917.
[3] Based on the publicly available report R5RS [24].

defined. In particular, we show how the SCHEME construct `call/cc` is precisely defined with only 3 equations. This methodology has been used for the last two years in a programming language design class taught by the second author at the University of Illinois [34]. Even though SCHEME is not concurrent in its core, the methodology we present also supports concurrency, as shown in [34, 27].

This paper is organized as follows. Section 2 describes related work. Section 3 introduces informally the MAUDE system and the SCHEME language. Section 5 shows the executable semantics of SCHEME. Section 5 illustrates performance results and Section 6 concludes the paper.

2 Related Work

Monads have been introduced [36] as a data type allowing one to add "impure" features, such as side-effects, to pure functional languages. They became notable for their contribution to modularity. Monads can be seen as a structuring feature that enables one to systematically write programs in tail-form. With monads it is easier to localize changes since context is transferred in a uniform manner. Mainly because of the use of continuations, transitions in our semantics are defined in a similar fashion as the monad operator $\star$. Matching modulo ACI allows one to define $\star$ via equations with some degree of freedom.

MSOS (*Modular Structural Operational Semantics*) [30] is a formalism which aims at describing modular definitions of languages. The technique in [30] proposes to overcome the possible lack of modularity in standard SOS [32] specifications. In MSOS, transitions take the form $\gamma \xrightarrow{\alpha} \gamma'$, where γ and γ' are called configurations and denote syntactic entities of the language (expression and commands, for instance). One can understand these transitions as describing source-to-source derivations rules. The label α transmits only the necessary information needed in a particular derivation, such as the store and the environment. Modularity is obtained essentially by avoiding the inclusion of auxiliary information explicitly in configuration terms. It is worth noting that MSOS [30, page 221] was influenced by monad transformers.

Braga *et al.* [11, 6, 8] investigated modularity of rewrite theories for defining programming languages. Meseguer and Braga [26] recently illustrated a technique, named *Modular Rewriting Semantics* (MRS) [26], for defining modular rewrite theories. They propose the use of *record inheritance*, and *abstract interfaces* in order to give *definitive* formalizations of language constructs via conservative extensions of the semantics. The first technique is similar to the notion of set of state attributes we use, which is essentially an AC list of attributes representing the "state of the program". The store, environment, and locks held by each thread are candidate attributes for this list in the semantics of concurrent languages. Matching modulo ACI makes possible to specify equations containing only a projection of the attributes in the list. Similarly to MSOS labels and MRS record inheritance, the use of sets of state attributes allows one to define only what is needed in order to carry out a derivation step, thus reducing the coupling between infrastructure and semantic definitions. The use of abstract interfaces as the second technique follows the information hiding discipline [31] which is a fundamental concept to achieve modularity. In our semantics, this boils down to defining separate modules for the syntax and semantics of each construct, and accessing modules via their public operations.

It is worth mentioning that in this paper we do *not* aim at formalizing our definitional methodology, or at comparing it mathematically to other existing techniques. Instead, our purpose here is to instantiate it to one non-trivial special case, the definition of SCHEME. We believe that the presentation will be self-explanatory and intuitive enough to make this possible.

3 Preliminaries

In this section we present a short introduction to the MAUDE language, then give an informal semantics to some SCHEME constructs.

Maude is a high-performance executable specification language for rewriting logics whose roots go back to CLEAR [7] and OBJ [19]. Many other languages belong to the same family, such as CAFEOBJ [13], BOBJ [18], and ELAN [5].

In MAUDE, specifications are introduced as theory modules. In this paper we only use functional modules, and for these, specifications correspond to equational theories of the form (Σ, E), where Σ denotes the signature of a module including its sorts and operations defining the interface to that module, and E denotes the set of equations that should hold for any implementation of that module. MAUDE replaces terms by terms when they are equal under an equational theory. That is, for any equation of the form $t_l = t_r$, when a term t can be matched modulo associativity, commutativity, and identity (ACI) to t_l via a substitution θ, then t can be replaced by t_r with variables substituted according to θ. As an example, the module below axiomatizes the PEANO natural numbers:

```
fmod PEANO-NAT is sort Nat .
  op zero : -> Nat .
  op succ : Nat -> Nat .
  op plus : Nat Nat -> Nat .
  vars N M : Nat .
  eq plus(zero, M) = M .
  eq plus(succ(N), M) = succ(plus(N, M)) .
endfm
```

The only sort defined in this module is `Nat`. The constant `zero` is defined as an operation with no arguments and has sort `Nat`. The operations `succ` and `plus` denote the successor of a number and the addition of two numbers, respectively. The equations in this module must hold for any model of `PEANO-NAT`. In these equations, variables are universally quantified. That is, the first equation describes the axiom: ∀ `M:Nat. 0 + M = M`. One can prove via equational deduction that the equality `plus(succ(succ(zero)),succ(succ(succ(zero)))) = plus(succ(succ(succ(succ(zero)))),succ(zero))` holds for this specification as both terms reduce to the ground term: `succ(succ(succ(succ(succ(zero)))))`.

In fact, one can show that this specification is confluent and terminates. One can import a module (theory) using the keyword `including`. All sorts, operations, and equations are then imported. The module `PEANO-NAT*` below extends the previous with multiplication.

```
fmod PEANO-NAT* is including PEANO-NAT .
  op mult : Nat Nat -> Nat .
  vars M N : Nat .
  eq mult(zero, M) = zero .
  eq mult(succ(N), M) = plus(mult(N, M), M) .
endfm
```

This simple examples illustrate most of MAUDE's features needed to define SCHEME. We explain other features by need. For a more complete description of MAUDE see [3].

SCHEME [24, 16] is a functional programming language mostly inspired by LISP. The following is a non-exhaustive list of its features:

– Statically scoped;
– Call-by-value (but can simulate call-by-need with the Promises data-type);
– Programs as data;
– Functions and continuations are first-class citizens;
– Dynamically typed.

SCHEME has a small core language. We next informally discuss its main constructs. The interested reader is referred to [24] for the standard definition and many examples. Notice in the following that, similarly to LISP, every expression in SCHEME has the syntax of a list.

Lists. The construct `list` has the form `(list exp1 ... expK)` and evaluates to a list of size k whose values correspond to the evaluation of each expression argument.

Variables. The binding construct `let` has the form

```
(let ((name1 exp1) ... (nameK expK)) exp1' ... expN')
```

and "atomically" binds the results of evaluating `exp1`,..., `expK` in the current environment to `name1`,...,`nameK`. The expressions `exp1'`,..., `expN'` are evaluated sequentially in the new environment obtained. The result of the whole expression is that of the last expression, `expN'`, evaluated in this sequence. Variations of this bindings include `let*` and `letrec`. The first performs the binding sequentially rather than atomically, while `letrec` binds in an environment initially extended with the name list `name1`,...,`nameK`. As usual, the `letrec` construct is very convenient for defining recursive definitions.

The construct `set!` has the form `(set! name exp)` and assigns the evaluation of `exp` to `name` in the current environment.

The construct `define` takes the form `(define name exp)` and binds the evaluation of `exp` to `name` in the top-level environment if `name` is not yet bound. Otherwise, this expression is similar to an assignment to `name`. Our interpreter does not allow the use of `define` outside the top-level block (it can be encoded with `letrec` [24]).

Functions. The function abstraction construct has the form

```
(lambda (name1 ... nameK) exp1' ... expN')
```

and evaluates to a closure. This closure saves the current environment because SCHEME is *statically scoped*. In addition, since the language supports *high-order* functions, closures are ordinary values in the language.

Function application takes the form `(exp0 exp1 ... expK)` where `exp0` must evaluate to a closure. Assuming that the closure that `exp0` evaluates to holds the function abstraction above and the environment $\mathcal{E}$, then the result of evaluating the application above is the same as that of evaluating

```
(let ((name1 exp1) ... (nameK expK)) exp1' ... expN')
```

in $\mathcal{E}$. Recall that SCHEME's parameter-passing style is *by-value*.

Control Flow. The `if` construct has the form `(if test then-expr else-expr)` and has the expected behavior of evaluating the expression `test` first and if the result is the "true" value then the result of the conditional is that of evaluating `then-expr`; otherwise that of evaluating `else-expr`.

Sequential composition has the form `(begin expr1`*c*`...exprK)` and evaluates sequentially each expression. The evaluation of `exprK` is the result of the entire expression.

Continuations are procedures of a single parameter and thus considered first-class citizens in SCHEME. In the example below, `k` denotes a *continuation procedure*. The actual parameter passed to the continuation procedure denotes the value that the *program continuation* expects to receive in order to continue. The way one creates a continuation procedure is by calling the procedure `call/cc` passing an expression denoting a function abstraction, which declares one formal standing for the continuation procedure. For instance, the SCHEME expression:

```
(let ((f (lambda (k) (k 10))))(* 5 (call/cc f)))
```

evaluates to the integer 50. When the function `f` is applied, a continuation procedure (that knows "how to continue") is created and passed as argument to `f`. The value `10` is then passed to the program continuation which is in charge of filling the hole □ in the expression `(* 5 □)` and evaluate it.

Data Types. In addition to function, continuation, and list data-types, SCHEME supports integer, rational, real, string, boolean, and several other data-types. We here discuss only the integers and booleans. Integer literals take the form `1, 2, 3, ...`, while the boolean literals are `#t` and `#f`. Arithmetic operations on these types are provided as built-in operations. For instance, `(+ 1 2)` denotes the addition of two integers and `(and a b)` denotes the conjunction of two expressions.

SCHEME also supports "frozen" expressions as values: expressions may evaluate to "frozen" expressions, also called "data". With the support of constructs to freeze and unfreeze "data", one can store programs as data. For instance, the construction `(quote exp)` freezes the expression `exp`. That means that `exp` is *not* evaluated when `(quote exp)` is. Conversely, the expression `(unquote exp)` evaluates the frozen expression that can be resulted from the evaluation of `exp` in the current environment.

Promises are introduced to support *call-by-need*. Two operations support this feature: `(delay exp)` and `(force exp)`. Similarly to quotations, the evaluation of `delay` does not trigger the evaluation of `exp`. It saves the current environment and `exp` in a promise data-type. When forced, the promise will evaluate `exp` in the saved environment. Additional calls to `force` on the same promise will return the same value. As one might expect, in the presence of *side-effects*, the evaluation of an expression can differ from an eventual force in the promise corresponding to that expression.

4 The SCHEMEM Syntax

We use a slightly different syntax in order to facilitate parsing and semantic definitions. We call this language SCHEMEM from here on. We write the syntax of SCHEMEM in MAUDE's mixfix notation which is equivalent to defining a context-free grammar. We next show fragments of the SCHEMEM syntax. Its entire syntax appears in [10]. We decided to have a single sort for expressions, namely `Exp`, in order to simplify the definition and parsing. The sorts, subsort relations, and operations below define the basic syntax for integers, booleans, and expression lists. The sorts `Qid`, standing for quoted identifiers, and `Int` are pre-defined by MAUDE. `NameList` is defined in a similar manner as `ExpList`. The `subsorts` keyword allows one to declare subsort relations succinctly. For instance, we declare `Qid` to be subsort of `Name`, and this to be a subsort of `Exp`:

```
sorts Exp ExpList .
subsorts Qid < Name Int < Exp NameList < ExpList .
subsort Name < NameList .
op a b c d e f g h i j k l m n o p q r s t u v x y z : -> Name .
op noExp : -> Exp . op _,_ : ExpList ExpList -> ExpList .
```

The operations `plus`, `minus`, `times`, `div`, and `mod` have signature `Exp Exp -> Exp`. Boolean expressions and the conditional are defined with the following syntax:

```
ops true false : -> Exp .
ops eq leq geq and or : Exp Exp -> Exp .
op not : Exp -> Exp .
op if_then_else_ : Exp Exp Exp -> Exp .
```

The syntax of `letrec` follows. The attribute `prec` assigns precedence to an operator in order to deal with parsing conflicts:

```
sorts Binding BindingList .  subsort Binding < BindingList .
op none : -> BindingList .
op _,_ : BindingList BindingList -> BindingList
         [assoc id: none prec 71] .
op _=_ : Name Exp -> Binding [prec 70] .
op letrec_in_ : BindingList Exp -> Exp .
```

The syntax of function abstraction and application is:

```
op lambda _ -> _ : NameList Exp -> Exp .
op __ : Exp ExpList -> Exp [prec 0] .
```

We define next the syntax for quotation, unquotation, callcc, and for the operations on the "promises" data type.

```
op quote : Exp -> Exp .
op unquote : Exp -> Exp .
op callcc_ : Exp -> Exp [prec 0] .
op delay : Exp -> Exp .
op force : Exp -> Exp .
```

Finally, we give the syntax for assignment, sequential composition, and expression blocks. Note that the sort `ExpList;` is used to stand out expressions separated by the `;` symbol.

```
op set_=_ : Name Exp -> Exp .
sort ExpList; .
subsort Exp < ExpList; .
op _;_ : ExpList; ExpList; -> ExpList; [assoc prec 100] .
op {_} : ExpList; -> Exp .
```

We implemented, but omitted in this paper for space reasons, the syntactic constructs for: vectors, lists, loops, `let*` , `let`, and IO operations.

5 The Equational Semantics

Next we show the state infrastructure used in the semantics, describe the continuation semantics, and finally instantiate this semantics to SchemeM.

5.1 The State Infrastructure

In what follows we axiomatize the standard notions of location, environment, value, and store. Each module denotes one relevant component of the state infrastructure, allowing us to eventually define the module `STATE`. We start with the formal definition of locations (term of sort `Location`), in the module `LOCATION` below. The sort of natural numbers, `Nat`, is imported from the built-in module `INT`. A list of locations can be constructed using the comma mixfix operator `_,_`. In MAUDE, one can assign certain properties to operators via a fixed set of attributes. For instance, we state that the operator `_,_` is commutative and `noLoc` is its identity. So the term `(loc(1),noLoc),loc(2)` is equivalent to `loc(1),loc(2)`. It is sometimes convenient to generate a fresh sequence of contiguous locations. This can be done with the operator `loc(_;;_)` taking as arguments two natural numbers: the index of the first location and the number of locations to create:

```
fmod LOCATION is including INT .
  sorts Location LocationList . subsort Location < LocationList .
  op loc : Nat -> Location .
  op noLoc : -> LocationList .
  op _,_: LocationList LocationList -> LocationList [assoc id: noLoc] .
  op loc : Nat -> Location .
  op loc(_;;_) : Nat Nat -> LocationList .
  vars N # : Nat .
  eq loc(N ;; 0) = noLoc .
  eq loc(N ;; #) = loc(N), loc((N + 1) ;; (# - 1)) .
endfm
```

Next we describe environments. The module `NAME-LIST` defines the sorts `Name` and `NameList` standing for the language identifier and its associated list (see [10]). An environment is a table mapping names to locations. An environment cell has the form `[N,L]` where `N` is a name and `L` a location. The operation `_ _` concatenates two environments. Note that, in addition to using the `assoc` and `id:` attributes, we also declare this operator commutative (`comm`). It is worth noting that these attributes are central to rewriting, since matching is performed modulo ACI (associativity, commutativity and identity). The operation `E[Nl<-Ll]` updates the environment `E` by binding the location list `Ll` to the name list `Nl` element-wise.

```
fmod ENVIRONMENT is including LOCATION . including NAME-LIST .
  sort Env .   op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .
  op _[_<-_] : Env NameList LocationList -> Env .
  vars X X' : Name .  vars Env Env' : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  eq Env[() <- noLoc] = Env .
  eq ([X,L] Env)[X,Xl <- L',Ll] = ([X,L'] Env)[Xl <- Ll] .
  eq Env[X,Xl <- L,Ll] = (Env [X,L])[Xl <- Ll] [owise] .
endfm
```

The next module, `CONTINUATION`, defines the sort `Continuation`, which is central to our continuation-based semantics. The constant `stop` is the empty continuation and denotes a point in which there is no continuation, the end of the program:

```
fmod CONTINUATION is
  sort Continuation .
  op stop : -> Continuation .
endfm
```

The following module, `VALUE`, defines the infrastructure for values; the sorts `Value` and `ValueList` denote the values and the list of values in the language. As usual, a closure value records the list of formals, the body expression, and an environment. It also stores the number of formals to avoid computing the number of formals in every application. The delayed value records to which location a delayed expression is bound, so that when forced we will be able to update that location with the computed value:

```
fmod VALUE is including ENVIRONMENT . including CONTINUATION .
  including GENERIC-EXP-SYNTAX .
  sorts Value ValueList .
  subsort Value < ValueList .
  op noVal : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noVal] .
  *** values of the language
  op noV : -> Value .
  op [_] : ValueList -> Value .
  op int : Int -> Value .
  op bool : Bool -> Value .
  op vector (_,_) : Location Nat -> Value .
  op frozen (_) : Exp -> Value .
  op closure : Nat NameList Exp Env -> Value .
  op delayed(_,_,_,_) : Exp Env Location Value -> Value .
  op kon(_) : Continuation -> Value .
endfm
```

A store is a table mapping locations to values, as the following module, called `STORE`, shows. Similarly to environments, a store cell has the form `[L,V]` for any location `L` and value `V`, and `__` is the operation that concatenates two stores. The update operation `_[_<-_]` on stores associates a location list to a value list element-wise.

```
fmod STORE is protecting LOCATION . including VALUE .
  including ENVIRONMENT . including GENERIC-EXP-SYNTAX .
  sort Store .  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
  op _[_<-_] : Store LocationList ValueList -> Store .
  ...
endfm
```

We finally define the module `STATE`, which puts together all the infrastructure defined above. A program state is a term having sort `MState`. It is comprised of a collection of state attributes concatenated with the operator `_,_`. The attribute `m` denotes the store of the program, which will be updated as the program runs. The attribute `o` contains a list of values that can be output when the program finishes.

The attribute `d` encloses the distinguished top-level environment updated by `define` expressions. The attribute `n` is the index of the next free location available in the store. Finally, the attribute `k` denotes the continuation of the program:

```
fmod STATE is including ENVIRONMENT .
  including STORE . including CONTINUATION .
  sorts StateAttribute MState . subsort StateAttribute < MState .
  op empty : -> MState .
  op _,_ : MState MState -> MState [assoc comm id: empty] .
  op m : Store -> StateAttribute .
  op o : ValueList -> StateAttribute .
  op d : Env -> StateAttribute .
  op n : Nat -> StateAttribute .
  op k : Continuation -> StateAttribute .
endfm
```

Next section uses this infrastructure in semantic definitions.

5.2 The Continuation Semantics

The operational semantics of a language is comprised of a set of axioms and inference rules [37, 20]. There are essentially two techniques for defining operational semantics: *structural* (a.k.a. small-step) due to Plotkin [32], and *natural* (a.k.a. big-step) due to Kahn [23]. Natural semantics is well known for its conciseness, since the transition relation can make big steps of evaluation, but is also known to be deficient in the support of concurrency, since it cannot capture intermediate interleavings of a program. Sequential languages are commonly defined using natural semantics [29]. Therefore, we show a simple *informal* translation of an inference rule given in big-step style to our semantics.

We take the IMP programming language defined in [37] as an example. In this definition, **Aexp** is the set of arithmetic expressions, **Loc** denotes the set of locations, $\mathbb{I}$ is the set of integer numbers denoting values[4], and Σ consists of the set of states, that is, functions σ: **Loc** $\to \mathbb{I}$. The transition relation $\to_{\mathcal{I}} \subseteq (\mathbf{Aexp} \times \Sigma) \times \mathbb{I}$ is defined inductively on the syntax of arithmetic expressions. The semantics of addition, for instance, can be defined by the inference rule:

$$\frac{(a_0, \sigma) \to_{\mathcal{I}} i \qquad (a_1, \sigma) \to_{\mathcal{I}} j}{(a_0 + a_1, \sigma) \to_{\mathcal{I}} n} \qquad \text{, where } n = i + j.$$

As shown in [30], unconstrained use of structural and natural semantics can lead to unmodular language formalizations. This means that the introduction of a new construct in the language may require modification in rules of other constructs, so one cannot hope that the definitions are definitive. As [30] shows, the undisciplined use of configurations mixing program and instrumental devices as the environment and store can lead to unmodular definitions.

[4] We changed the set of values to be integers to make the examples uniform with our infrastructure.

We next *informally* instantiate the CPS (*Continuation Passing Style*) semantics [34, 27] for the self-contained example of addition. The CPS semantics represents memory states as an (AC) set of attributes whose sort is `MState` as declared in the state infrastructure. This allows one to declare only what is needed, i.e. a projection of the state, in derivations and therefore reduce coupling between definitions and the infrastructure. Furthermore, the CPS semantics is based on continuations. This makes the definition of control-flow extremely simple and provides a systematic way for describing the semantics. The use of continuations is pervasive in programming languages [21, 16, 4] and for all purposes here we can consider them just as a stack. The dynamic semantics therefore retrieves and pushes continuation items from and to the top of such stack.

In general, the semantics of each language construct can be defined with two equations following a *divide-and-conquer* methodology. The first equation breaks down an expression into sub-expressions and puts them back on the continuation followed by a mark used to remember what need to be done to complete the evaluation of the original expression. The second equation is in charge of collecting the parts, removing the mark, and computing the value denoting the evaluation of the original expression.

In CPS [27], addition of two expressions a_0 and a_1 can be defined as follows:

$\vdash$ `k([plus(`a_0`, `a_1`) @ Env] -> K)` $=$ `k((`a_0`, `a_1` @ Env) -> {+} -> K)`
$\vdash$ `k([int(i),int(j)] -> {+} -> K)` $=$ `k(int(i+j) -> K)`

Note that many of these terms are yet to be defined. However, this simple example gives an evidence that the transition relation $\rightarrow_{\mathcal{I}}$ can be implemented as a term rewrite system. In fact, these axioms are implemented as *unconditional* equations. Note also that we do not need to mention all the attributes in the state. We just specify what we need in order for a derivation step to take place. Here, for instance, we do not mention the store. The variable `K` stands for the continuation that should be followed after evaluating `plus(`a_0`, `a_1`)` at environment `Env`. Note that after evaluating the addition in the domain of integers, the resulting value will be available on the top of the continuation so another "computation" can proceed independently. The following is a non-exhaustive list of features of the CPS semantics:

- it uses an AC list of *state attributes*;
- it is defined in a continuation passing style;
- expressions carry their evaluation environment.

The first and second design decisions aim at increasing modularity. The second is also central to the definition of control sensitive features, and the third avoids the tedious task of recovering the environment when leaving blocks.

As we illustrate in the next section, the ***run*** of a program `E` is a sequence of the form $s_0 \rightsquigarrow s_1 \rightsquigarrow \cdots \rightsquigarrow s_n$, where $\rightsquigarrow$ stands for term rewrites, s_i (for $0 \leq i \leq n$) has sort `MState`, s_0 = `(k([E @ noEnv] -> stop), n(0),d(noEnv),m(noStore),o(noVal))` is the initial state, and s_n is a state containing the continuation `k(V -> stop)`. The term `[E @ noEnv] -> stop` occurring in s_0 denotes the current (initial) continuation. It "says" that the expression `E` waits to be evaluated in the environment `noEnv`, and nothing else remains to be evaluated. One can understand the equational theory we define as the definition of an abstract state machine implementing the natural semantics of Scheme.

Note about concurrency

Our CPS definitional methodology informally presented above is capable of translating big-step declarations *without* compromising concurrent behavior. This perhaps surprising claim [30, page 216] relies on the fact that rewrite rules are applied *modulo* equations in a rewriting engine like Maude. Operationally, equations are applied first, exhaustively in order to "cannonicize" the term/ state to rewrite, then one rewrite rule is applied, then equations again, etc. In order to preserve concurrent semantics, the right notion of thread/process interference needs to be captured using rewrite rules.

One needs to identify which operations affect concurrent behavior. For non-concurrent operations, the "state (term) graph" will commute and therefore the intermediate states do not need to be observable. This technique essentially corresponds to a partial-order reduction [17] on the language semantics. In the case of a language like Java [14], only three operations are treated as concurrent: read and write of shared variables, and acquisition of locks. Instead of equations, the semantics of these three operations are defined as rewrite rules in Maude using `rl` instead of `eq`. Except for these three cases, every other operation is defined equationally. The small number of rewrite rules improves considerably the performance of *state search*, as JavaFAN demonstrates [14].

Of course, one still needs to increment the infrastructure to support concurrency. Instead of keeping only one continuation in the memory state, one needs to create as many continuations as threads spawned and associate to each thread the set of locks that it holds.

Even though our methodology works well for concurrent language definitions, in this paper we only show it at work on a sequential language definition. However, we believe that Scheme, and specially its `call/cc` feature, is complex enough to reflect the strength of our CPS definitional methodology.

5.3 Maude Functional Modules

Due to space limitations we describe only one important subset of modules in the definition of SchemeM. The complete definition is available at [1].

We first present the module `GENERIC-EXP-K-SEMANTICS`. The continuation `[_@_]->_` is used throughout the semantics to represent expressions at a given environment. Note that the evaluation of an integer literal is the corresponding integer value. The evaluation of a name, however, needs the store in order to retrieve the associated value. Note that only the continuation, top-level environment, and store attributes "`k, d, m`" are used in that definition. The top-level environment is searched if the name can not be reached from the current environment. The attribute `owise` is essentially giving precedence to the previously defined equation. The continuation `[_@_|_]->_` is used to define the evaluation of an expression list into a value list. Finally, a store update occurs when a list of values appears in front of a list of locations on the top of the continuation:

```
fmod GENERIC-EXP-K-SEMANTICS is including GENERIC-EXP-SYNTAX .
  including EXP-LIST . including STATE . including VALUE .
  op [_@_] ->_: ExpList Env Continuation -> Continuation .
  op [_@_|_] ->_: ExpList Env ValueList Continuation -> Continuation .
  op _->_ : ValueList Continuation -> Continuation .
  op _->_ : LocationList Continuation -> Continuation .
  vars E E' : Exp . var El : ExpList . var V : Value .
  var Vl : ValueList . var L : Location . var Ll : LocationList .
  var X : Name . var Xl : NameList .  var S : MState . var I : Int .
  var K : Continuation .  var M : Store . vars Env Env' : Env .
  eq k([I @ Env] -> K) = k(int(I) -> K) .
  *** name expression
  eq k([X @ Env[X,L]] -> K), d(Env'), m([L,V] M)
   = k(V -> K), d(Env'),  m([L,V] M) .
  *** look for top-level definitions
  eq k([X @ Env] -> K) , d([X,L] Env') , m([L,V] M)
   = k(V -> K) , d([X,L] Env') , m([L,V] M) [owise] .
  eq k([(E,E',El) @ Env] -> K)
   = k([E @ Env] -> [(E',El) @ Env | noVal] -> K) .
  *** atomic memory block write; useful for let and letrec
  eq k(V -> [() @ Env | Vl] -> K) = k((Vl,V) -> K) .
  eq k(V -> [(E,El) @ Env | Vl] -> K)
   = k([E @ Env] -> [El @ Env | Vl,V] -> K) .
  eq k(Vl -> Ll -> K) , m(M) = k(K) , m(M[Ll <- Vl]) .
  eq k([ noExp @ Env ] -> K) = k(K) .
endfm
```

As usual, in the semantics of `if-then-else` we first evaluate the boolean expression and, depending on the resulted value, we evaluate the `then` expression or the `else`. Note that we rely on the definition of the MAUDE `if-then-else` in the last equation:

```
fmod IF-K-SEMANTICS is including IF-SYNTAX .
  including BEXP-K-SEMANTICS .
  op [if(_,_) @_] -> _ :
     Exp Exp Env Continuation -> Continuation .
  vars BE E E' : Exp .  var B : Bool .
  var K : Continuation .  var Env : Env .
  eq k([ if BE then E else E' @ Env] -> K)
   = k([BE @ Env] -> [if(E,E') @ Env] -> K) .
  eq k(bool(B) -> [if(E,E') @ Env] -> K)
   = k(if B then [E @ Env] -> K else [E' @ Env] -> K fi) .
endfm
```

The following module, BINDING-K-SEMANTICS, is needed for the definition of binding expressions. The operator a has the sole purpose of separating name and expression in a binding list and also of indicating the number of bindings in that list:

```
fmod BINDING-K-SEMANTICS is including BINDING-SYNTAX .
  including GENERIC-EXP-K-SEMANTICS . sort Aux .
  op a : Nat NameList ExpList BindingList -> Aux .
  var N : Nat . var Xl : NameList .  var El : ExpList .
  var X : Name .  var E : Exp .  var Bl : BindingList .
  eq a(N, Xl ,El, (X = E, Bl)) = a(N + 1, (Xl,X), (El,E), Bl) .
endfm
```

The construct letrec, defined in the next module, first creates a new environment containing all the names declared, and then evaluate the expressions in the binding list. Finally, the binding is performed and the body evaluated in this new environment. The attribute := in the conditional performs a pattern-matching when the term on the right cannot be further reduced. We could remove this conditional equation by passing the term a to the continuation. We decided to keep it for the sake of readability:

```
fmod LETREC-K-SEMANTICS is including LETREC-SYNTAX .
  including BINDING-K-SEMANTICS .
  op letrec : Nat NameList ExpList Exp -> Exp . vars N # : Nat .
  var Bl : BindingList . var E : Exp .  var Xl : NameList .
  var El : ExpList . var K : Continuation .  var Env : Env .
 ceq letrec Bl in E
    = letrec(#,Xl,El,E) if a(#,Xl,El,none) := a(0,(),(),Bl) .
  eq k([letrec(#,Xl,El,E) @ Env] -> K) , n(N)
    = k([El @ Env[Xl <- loc(N ;; #)]] -> loc(N ;; #) ->
      [E @ Env[Xl <- loc(N ;; #)]] -> K) , n(N + #) .
endfm
```

The module PROC-K-SEMANTICS below defines the semantics of procedures. The operator function is a convenience to avoid computing the number of formals for every call of the same function. Note that the equation for this operator performs a source-to-source transformation on the program (term). The operation length, defined in the syntactic module NAME-LIST, computes the size of a name list. The function abstraction evaluates to a closure, which is a value of the language (it can be passed around or stored) that also saves the environment (recall that SCHEME is statically-scoped). Function application is defined with two equations. The first evaluates the expressions standing for the closure and actual parameters. The second binds the actual parameters to their formals in the environment saved in the closure. Then, the body is evaluated in the resulting context:

```
fmod PROC-K-SEMANTICS is including PROC-SYNTAX .
  including GENERIC-EXP-K-SEMANTICS .
  op function : Nat NameList Exp -> Exp .
  op fn -> _ : Continuation -> Continuation .
  var Xl : NameList .  var El : ExpList .  var F E : Exp .
  var K : Continuation .  var Env : Env .  var M : Store .
  vars N # : Nat .  var Vl : ValueList .
  *** abstraction
  eq lambda(Xl) -> E = function(length(Xl),Xl,E) .
  eq k([function(#,Xl,E)@Env] -> K) = k(closure(#,Xl,E,Env) -> K) .
  *** application
  eq k([(F(El)) @ Env] -> K) = k([(F,El) @ Env] -> fn -> K) .
  eq k((closure(#,Xl,E,Env), Vl) -> fn -> K) , n(N), m(M)
    = k([E @ Env[Xl <- loc(N;;#)]] -> K) , n(N + #),
      m(M[loc(N;;#) <- Vl]) .
endfm
```

The next module illustrates the semantics of the constructs `quote` and `unquote`. The expression `quote` casts a program fragment to data. It means that the enclosed expression will not be evaluated when `quote` is. A frozen value includes solely the string of text for a program. The expression `unquote` takes a `frozen` value and evaluates it in the current context (Scheme provides other constructs to facilitate the composition of program and data – see `quasiquotation` in [24] – in lists, but we do not discuss these here):

```
fmod LITERAL-K-SEMANTICS is including LITERAL-SYNTAX .
  including BEXP-K-SEMANTICS . including PROC-K-SEMANTICS .
  including AEXP-K-SEMANTICS . including ENVIRONMENT .
  var K : Continuation . var Xl : NameList . var Vl : ValueList .
  var E : Exp . vars Env Env' : Env . vars N : Nat . var V : Value .
  op [eval_] -> _ : Env Continuation -> Continuation .
  eq k([quote(E) @ Env] -> K) = k(frozen(E) -> K) .
  eq k([unquote(E) @ Env] -> K) = k([E @ Env] -> [eval(Env)] -> K) .
  eq k(frozen(E) -> [eval(Env)] -> K) = k([E @ Env] -> K) .
  eq k([closure(N , Xl , E , Env) , Vl] -> [eval(Env')] -> K)
    = k((closure(N,Xl,E,Env), Vl) -> fn -> K) .
  eq k(V -> [eval(Env)] -> K) = k(V -> K) [owise] .
endfm
```

Scheme's `call/cc` is perhaps the most difficult to understand feature of this language. Despite its complexity it can be defined very concisely in our semantics. This happens mainly because we encode explicitly the continuation in our specification. Note that we introduce a new value, `kon(_)`, which encloses a continuation. Thus, continuations are regarded first-class citizens; it can be passed around and stored. The expression `callcc(_)` is a special kind of function call. When this expression is evaluated, it triggers a call to the function denoted by its parameter, passing the current continuation as a value to this function. Further, this continuation can be "called" like any other function, as the last equation shows. When this happens, the stored continuation replaces the current and the actual parameters are placed in the top of the new continuation:

```
fmod CALLCC-K-SEMANTICS is including CALLCC-SYNTAX .
  including AEXP-K-SEMANTICS . including GENERIC-EXP-K-SEMANTICS .
  including PROC-K-SEMANTICS . var E : Exp .
  var Xl : NameList . var X : Name . vars K K' : Continuation .
  vars Env Env' : Env . var Vl : ValueList . var # : Nat .
  op kon(_) : Continuation -> Value .
  op [callcc] ->_ : Continuation -> Continuation .
  eq k([(callcc E) @ Env] -> K) = k([E @ Env] -> [callcc] -> K) .
  eq k((closure(#,Xl,E,Env)) -> [callcc] -> K)
    = k(kon(K) -> [() @ noEnv | closure(#,Xl,E,Env)] -> fn -> K) .
  eq k((kon(K),Vl) -> fn -> K') =  k(Vl -> K) .
endfm
```

The module `VAR-ASSIGNMENT-K-SEMANTICS` below is self- explanatory:

```
fmod VAR-ASSIGNMENT-K-SEMANTICS is including VAR-ASSIGNMENT-SYNTAX .
  including GENERIC-EXP-K-SEMANTICS .
  var X : Name . var E : Exp . vars Env Env' : Env . var M : Store .
  var K : Continuation .  var L : Location .  var V : Value .
  op _=>_ : Location Continuation -> Continuation .
  eq k([(set X = E) @ ([X,L] Env)] -> K)
   = k([E @ ([X,L] Env)] -> L => int(1) -> K) .
  eq k([(set X = E) @ (Env)] -> K) , d([X,L] Env')
    = k([E @ (Env)] -> L => int(1) -> K) , d([X,L] Env') .
  eq k(V -> L => K) , m(M) = k(K) , m(M[L <- V]) .
endfm
```

The next module, `BLOCK-K-SEMANTICS`, defines sequential composition of expressions:

```
fmod BLOCK-K-SEMANTICS is including BLOCK-SYNTAX .
  including GENERIC-EXP-K-SEMANTICS .
  op ignore -> _ : Continuation -> Continuation .
  var E : Exp . var El; : ExpList; .
  var Env : Env . var K : Continuation .  var V : Value .
  eq k([{E} @ Env] -> K) = k([E @ Env] -> K) .
  eq k([{E ; El;} @ Env] -> K)
   = k([E @ Env] -> ignore -> ([{El;} @ Env] -> K)) .
  eq k(V -> ignore -> K) = k(K) .
  eq k(ignore -> K) = k(K) .
endfm
```

The module `DELAYED-EXPRESSION-K-SEMANTICS` implements the "promises" datatype. A promise results from the evaluation of a `delay(_)` expression. In contrast to `quote`, this expression stores the current environment which will be used to evaluate the stored expression in the future. This data-type can be used to model *call-by-need*. The construct `force(_)` is used to reclaim the promise. Note that if such value is stored at some location, then the result of the evaluation will be memorized at the same location. This avoids the expression to be evaluated several times:

```
fmod DELAYED-EXPRESSION-K-SEMANTICS is including PL-SYNTAX .
  including DELAYED-EXPRESSION . including GENERIC-EXP-K-SEMANTICS .
  var E : Exp . var Env : Env . var V : Value .
  var K : Continuation . var L : Location . var M : Store .
  op [force] -> _ : Continuation -> Continuation .
  op [memo_] -> _ : Value Continuation -> Continuation .
  eq k([delay(E) @ Env] -> K) = k(delayed(E, Env , noL , noV) -> K) .
  eq k([force(E) @ Env] -> K) = k ( [E @ Env] -> [force] -> K) .
  eq k(delayed(E,Env,noL,noV) -> [force] -> K) = k([E @ Env] -> K) .
  eq k(delayed(E,Env,L,noV) -> [force] -> K)
    = k([E @ Env] -> [memo( delayed(E,Env,L,noV) )] -> K) .
  --- if the promise is evaluated
  eq k(delayed(E,Env,L,V) -> [force] -> K) = k (V -> K) [owise] .
  eq k(V -> [memo(delayed(E,Env,L,noV))] -> K) , m(M)
    = k(V -> K) , m(M[L <- delayed(E,Env,L,V)]) .
endfm
```

The semantics of SchemeM is obtained by including the semantics of the features discussed above (and others). We define an operator `eval` in terms of a set of state attributes, denoting the "initial state". In this set we add all the state attributes required to perform an evaluation, and include the expression (the program) on the top of the continuation, with an empty environment. The evaluation terminates when the stop continuation is reached:

```
fmod PL-K-SEMANTICS is protecting PL-SYNTAX .
  including AEXP-K-SEMANTICS . including BEXP-K-SEMANTICS .
  including VECTOR-K-SEMANTICS . including LIST-K-SEMANTICS .
  including IF-K-SEMANTICS . including LET-STAR-SEMANTICS .
  including LET-K-SEMANTICS . including PROC-K-SEMANTICS .
  including LITERAL-K-SEMANTICS . including CALLCC-K-SEMANTICS .
  including LETREC-K-SEMANTICS .
  including VAR-ASSIGNMENT-K-SEMANTICS .
  including BLOCK-K-SEMANTICS . including LOOP-K-SEMANTICS .
  including CASE-K-SEMANTICS . including IO-K-SEMANTICS .
  including DEFINE-K-SEMANTICS .
  including DELAYED-EXPRESSION-K-SEMANTICS .
  op eval : Exp -> Value .   op [_] : MState -> Value .
  var V : Value .  var S : MState .  var I : Int .
  var K : Continuation . var Vl : ValueList . var X : Name .
  eq eval(E)
   = [(k([E @ noEnv] -> stop)), n(1), m(noStore), o(noVal), d(noEnv)] .
  eq [k(V -> stop), S] = V .
  eq [k(stop), S] = noVal .
endfm
```

6 Performance Results

Figures 1 to 5 illustrate programs from a benchmark we used to evaluate the performance of our interpreter. Figures 1 to 3 show different implementations of the factorial function. Figure 3, for instance, shows the factorial program implemented with `callcc`. Figure 4 shows insertion sort and Figure 5 illustrates a program calculating all the permutations up to a natural number n. We created empty functional modules declaring constant names standing for the names used throughout these programs in order to avoid the need of using quotation marks.

For instance, we declared `fact-recursive` , `fact-iterative` as constant names inside a module including the specification of our interpreter.

```
{ define x = 10000 ;
  letrec fact-recursive =
     lambda (n) -> if eq(n , 1)
                   then 1
                   else times(n ,
                              fact-recursive(minus(n , 1)))
  in (f x)  }
```

Figure 1: Recursive factorial.

```
{ define x = 50000 ;
  let fact-iterative = lambda (n) ->
     let p = 1 , i = 2 in {
        while leq(i , n) {
           set p = times( p , i ) ;
           set i = plus( i , 1 )
        } ;
        p
     }
  in fact-iterative(x)
}
```

Figure 2: Iterative factorial.

Table 1 shows the running times and the number of rewrites (inside parenthesis) of various programs in a benchmark comparing our implementation of SchemeM with that of Dr.Scheme, a well-known C implementation of Scheme. We used a two 2.4GHz, 512KB cache processor machine with 4GB of memory running Fedora Core release 1. The SchemeM programs are listed in [10]. Note that the running times of the first four entries are close to that of Dr. Scheme. We believe that the highly efficient implementation of lists in Dr. Scheme might be the reason for the difference in the last 2 entries. We included in this table entries to the recursive and iterative factorial of 300. These were also used in [8], where they achieved results of 0.45 seconds and 23,122 rewrites, and 0.52 seconds and 33,511 rewrites for the recursive and iterative implementations, respectively. Note, however, that [8] implements a different language using a different methodology.

It is expected that *compilation* of equational specifications can lead to significant speed-ups. We hope that in the foreseeable future, our approach can

```
let g =
  lambda ('v1,'v2) ->
    let r = lambda ('kon) -> list ('kon , 'v1 , 'v2)
    in callcc(r)
in let 'fact-callcc =
     lambda (v) ->
       let 'triple = g(1,v)
       in let 'first = car('triple),
              'second=car(cdr('triple)),
              'third=car(cdr(cdr('triple)))
          in {
            if eq('third,0)
            then 'second
            else 'first (
                  list('first , times ('second , 'third ) , minus('third , 1 ) )
                 )
          }
   in 'fact-callcc(25000)
```

Figure 3: Factorial via callcc.

```
letrec
  insert = lambda (x,l) ->
                 if null?(l)
                 then list(x)
                 else if leq( x , car(l))
                      then cons(x,l)
                      else cons(car(l), insert(x,cdr(l)))
  , insert-sort = lambda (l) ->
                 if null?(l)
                 then emptyList
                 else insert(car(l), insert-sort(cdr(l)))
  , genlist = lambda (n) ->
                 if eq(n , 0)
                 then emptyList
                 else cons( n , genlist( minus(n,1) ) )
in insert-sort ( genlist(200) )
```

Figure 4: Insertion sort.

not only serve to prototyping and analyzing programs, but also as a way to deliver very efficient language interpreters. The source code and benchmarks of SchemeM can be downloaded at [1].

7 Conclusions

This paper presents the semantics of the language Scheme (R5RS) as an equational theory in the Maude rewriting system. We claim that the definition is highly modular and thus can be easily extended with new constructs. Similar semantics for Java, JVM, OCamL, ML, and Haskel [2, 14] have already been or are currently being developed in the Formal Systems Laboratory at the University of Illinois at Urbana-Champaign.

```
letrec
    permutations-up-to-n = lambda (n) ->
      if eq (n , 0)
      then list(emptyList)
      else insert(n, permutations-up-to-n( minus(n , 1) ))
, insert = lambda (n,l) ->
      if null?(l) then emptyList
      else app(interleave(n,car(l)), insert(n,cdr(l)))
, interleave = lambda (n,l) ->
      if null?(l) then list( list(n) )
      else cons(cons(n,l), mycons(car(l),
                            interleave(n,cdr(l))))
, mycons = lambda (x,l) ->
      if null?(l)
      then emptyList
      else cons(cons(x,car(l)), mycons(x,cdr(l)))
, app = lambda (u,v) ->
      if null?(u) then v else cons(car(u), app( cdr(u), v ))
in permutations-up-to-n(7)
```

Figure 5: List of $n!$ permutations over the first n natural numbers.

Program	SchemeM		Dr.Scheme mzScheme mode
`fact-recursive(300)`	0.040	(12,319)	0.000
`fact-recursive(25000)`	9.170	(1,025,019)	4.547
`fact-iterative(300)`	0.030	(11,749)	0.010
`fact-iterative(25000)`	7.440	(975,049)	6.299
`fact-callcc(25000)`	15.490	(2,100,197)	9.591
`insert-sort(400)`	43.840	(4,269,102)	0.110
`permutations-up-to-n(8)`	112.770	(12,776,619)	0.421

Table 1: Running time in seconds for a benchmark.

We strongly believe that these efforts might also set the ground for the development of software analysis tools. Maude already provides the user with a breadth-first search and LTL model-checker for theories developed in it. Recently, our group implemented dimensional analysis of C programs [35]. In [34], students defined the Hindley-Milner W [28] type inference algorithm in a two-week assignment. We currently investigate the use of equational logics in the development of static analyzers using some of the techniques presented in this paper.. In particular, the definition of symbolic execution tools, abstract interpreters [15, 22], and verification condition generators with their automatic, instead of interactive, theorem provers [12, 33] for domain-specific logics.

References

1. Equational Semantics of Scheme. http://fsl.cs.uiuc.edu/es/scheme.
2. Formal Systems Laboratory. http://fsl.cs.uiuc.edu.
3. Maude website. http://maude.cs.uiuc.edu/.
4. A. W. Appel and T. Jim. Continuation-Passing, Closure-Passing Style. In *Proceedings of POPL '89*, pages 293–302. ACM Press, 1989.

5. P. Borovansky, C. Kirchner, H. Kirchner, and P. E. Moreau. ELAN from a Rewriting Logic Point of View. *Theoretical Computer Science*, 285(2):155–185, 2002.
6. C. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Mapping Modular SOS to Rewriting Logic. In *Proceedings of 12th LOPSTR*, volume 2664 of *LNCS*, pages 262–277. Springer, 2003.
7. R. M. Burstall and J. A. Goguen. The Semantics of CLEAR, A Specification Language. In *Proceedings of the Abstract Software Specifications, 1979 Copenhagen Winter School*, pages 292–332, London, UK, 1980. Springer-Verlag.
8. F. Chalub and C. Braga. A Modular Rewriting Semantics for CML. *Journal of Universal Computer Science*, 10(7):789–807, July 2004. `http://www.jucs.org/jucs_10_7/a_modular_rewriting_semantics`.
9. M. d'Amorim and G. Roşu. An Equational Specification for the Scheme Language. In *Proceedings of SBLP'05*, pages 229–242, Recife, Pernambuco, Brazil, May 2005. Universidade de Pernambuco Press.
10. M. d'Amorim and G. Roşu. An Equational Specification for the Scheme Language. Technical Report UIUCDCS-R-2005-2567, University of Illinois Urbana-Champaign, April 2005.
11. C. de Oliveira Braga. *Rewriting Logic as a Semantic Framework for Modular Operational Semantics*. PhD thesis, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 2001.
12. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a Theorem Prover for Program Checking. *Journal of the ACM*, 52(3):365–473, 2005.
13. R. Diaconescu and K. Futatsugi. CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. *AMAST Series in Computing*, 1998. volume 6.
14. A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal Analysis of Java Programs in JavaFAN. In *CAV*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004.
15. B. Fischer and G. Roşu. Interpreting Abstract Interpretations in Membership Equational Logic. *Electronic Notes in Theoretical Computer Science*, 59(4):271–285, 2001.
16. D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw Hill, 1992.
17. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., 1996.
18. J. Goguen, K. Lin, G. Rosu, A. Mori, and B. Warinschi. An Overview of the Tatami Project. In T. T. Kokichi Futatsugi and A. Nakagawa, editors, *Cafe: An Industrial-Strength Algebraic Formal Method*, pages 61–78. Elsevier, 2000.
19. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. P. Jouannaud. Introducing OBJ. In J. Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.
20. C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.
21. R. Harper. *Programming Languages: Theory and Practice*. draft, 2004. http://www-2.cs.cmu.edu/~rwh/.
22. N. D. Jones and F. Nielson. Abstract Interpretation: a Semantics-Based Tool for Program Analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.
23. G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences*, pages 22–39. Springer-Verlag, 1987.
24. R. Kelsey, W. Clinger, and J. R. (Editors). Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
25. N. Martí-Oliet and J. Meseguer. Rewriting Logic as a Logical and Semantic Framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

26. J. Meseguer and C. Braga. Modular Rewriting Semantics of Programming Languages. In *Proceedings of the 10th International Conference, AMAST'04*, pages 364–378. LNCS, 2004.
27. J. Meseguer and G. Roşu. Rewriting Logic Semantics: From Language Specifications to Formal Analysis Tools . In *Proceedings of the IJCAR 2004*. LNCS, 2004.
28. R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, Aug. 1978.
29. R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
30. P. D. Mosses. Modular Structural Operational Semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.
31. D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.
32. G. D. Plotkin. Lecture notes DAIMI FN-19: A Structural Approach to Operational Semantics. Dept. of Computer Science, University of Aarhus, 1981.
33. A. Riazanov and A. Voronkov. The Design and Implementation of VAMPIRE. *AI Communications*, 15(2):91–110, 2002.
34. G. Roşu. Lecture notes of course on Programming Language Design. Dept. of Computer Science, University of Illinois U.C., 2005. http://www-courses.cs.uiuc.edu/˜cs422.
35. G. Roşu and F. Chen. Certifying Measurement Unit Safety Policy. In *Proceedings of the ASE'03*, pages 304–309, 2003.
36. P. Wadler. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.
37. G. Winskel. *The Formal Semantics of Programming Languages: an Introduction*. MIT Press, 1993.